PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/46 — A1

(11) International Publication Number: WO 98/02809

(43) International Publication Date: 22 January 1998 (22.01.98)

(21) International Application Number: PCT/US97/11879

(22) International Filing Date: 10 July 1997 (10.07.97)

(30) Priority Data:
08/680,202 11 July 1996 (11.07.96) US

(71) Applicant: TANDEM COMPUTERS INCORPORATED [US/US]; 10435 N. Tantau Avenue, LOC. 200-16, Cupertino, CA 95014 (US).

(72) Inventor: SCHOFIELD, Andrew; Lindenbuehl 27, CH-6330 Cham (CH).

(74) Agents: GRANATELLI, Lawrence, W. et al.; Graham & James LLP, 600 Hansen Way, Palo Alto, CA 94304 (US).

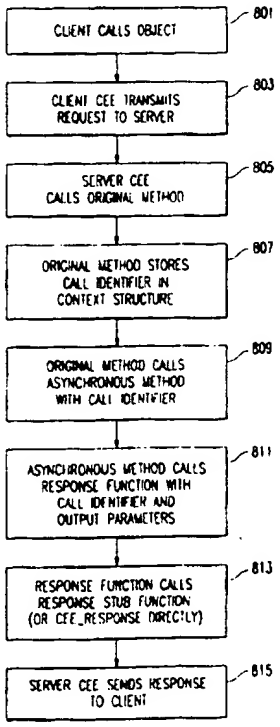
(81) Designated States: JP, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: METHOD AND APPARATUS FOR ASYNCHRONOUSLY CALLING AND IMPLEMENTING OBJECTS

(57) Abstract

A method and apparatus for asynchronously calling and implementing objects is disclosed. Object calls to perform an operation are performed asynchronously by calling the appropriate stub function from the client application and passing in the object reference, input parameters, and a pointer to a completion routine. The object reference, input parameters, and completion routine address are provided to a client-side execution environment. The client-side execution environment stores the completion routine address and transmits the input parameters to a server-side execution environment. The server-side execution environment calls a method in the server application that implements the requested operation. The server application performs the requested operation. Once the call has been responded to, the client-side execution environment calls the completion routine and passes it the output parameters to the requested operation. The client application can continue performing other asynchronous operations before the completion routine is called. To asynchronously implement an object that has been called, the appropriate method function in the server application is called which, in turn, calls an asynchronous operation. Once the asynchronous operation returns, the application responds to the client application.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania
AM	Armenia
AT	Austria
AU	Australia
AZ	Azerbaijan
BA	Bosnia and Herzegovina
BB	Barbados
BE	Belgium
BF	Burkina Faso
BG	Bulgaria
BJ	Benin
BR	Brazil
BY	Belarus
CA	Canada
CF	Central African Republic
CG	Congo
CH	Switzerland
CI	Côte d'Ivoire
CM	Cameroon
CN	China
CU	Cuba
CZ	Czech Republic
DE	Germany
DK	Denmark
EE	Estonia
ES	Spain
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
GE	Georgia
GH	Ghana
GN	Guinea
GR	Greece
HU	Hungary
IE	Ireland
IL	Israel
IS	Iceland
IT	Italy
JP	Japan
KE	Kenya
KG	Kyrgyzstan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
KZ	Kazakstan
LC	Saint Lucia
LI	Liechtenstein
LK	Sri Lanka
LR	Liberia
LS	Lesotho
LT	Lithuania
LU	Luxembourg
LV	Latvia
MC	Monaco
MD	Republic of Moldova
MG	Madagascar
MK	The former Yugoslav Republic of Macedonia
ML	Mali
MN	Mongolia
MR	Mauritania
MW	Malawi
MX	Mexico
NE	Niger
NL	Netherlands
NO	Norway
NZ	New Zealand
PL	Poland
PT	Portugal
RO	Romania
RU	Russian Federation
SD	Sudan
SE	Sweden
SG	Singapore
SI	Slovenia
SK	Slovakia
SN	Senegal
SZ	Swaziland
TD	Chad
TG	Togo
TJ	Tajikistan
TM	Turkmenistan
TR	Turkey
TT	Trinidad and Tobago
UA	Ukraine
UG	Uganda
US	United States of America
UZ	Uzbekistan
VN	Viet Nam
YU	Yugoslavia
ZW	Zimbabwe

METHOD AND APPARATUS FOR ASYNCHRONOUSLY CALLING AND IMPLEMENTING OBJECTS

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a method and apparatus for making asynchronous object calls and asynchronous object implementations in client applications and server applications, respectively.

2. Background

Distributed object computing combines the concepts of distributed computing and object-oriented computing. Distributed computing consists of two or more pieces of software sharing information with each other. These two pieces of software could be running on the same computer or on different computers connected to a common network. Most distributed computing is based on a client/server model. With the client/server model, two major types of software are used: client software, which requests the information or service, and server software, which provides or implements the information or service.

Object-oriented computing is based upon the object model where pieces of code called "objects"--often abstracted from real objects in the real world--contain data (called "attributes" in object-oriented programming parlance) and may have actions (also known as "operations") performed on it. An object is defined by its interface (or "class" in C++ parlance). The interface defines the characteristics and behavior of a kind of object, including the operations that can be performed on objects of that interface and the parameters to that operation. A specific instance of an object is identified within a distributed object system by a unique identifier called an object reference.

In a distributed object system, a client application sends a request (or "object call") to a server application. The request contains an indication of the operation to be performed on a specific object, the parameters to that operation, the object reference for that object, and a mechanism to return error information (or "exception information") about the success or failure

of a request. The server application receives the request and carries out the request via a server "implementation." The implementation satisfies the client's request for an operation on a specific object. The implementation includes one or more methods, which are the portions of code in the server application that actually do the work requested of the implementation. If the implementation is carried out successfully, the server application returns a response to the client, if necessary. The server application may also return exception information.

To standardize distributed object systems, the Object Management Group ("OMG"), a consortium of computer software companies, proposed the Common Object Request Broker Architecture ("CORBA"). Under the CORBA standard, an Object Request Broker ("ORB") provides a communication hub for all objects in the system passing the request to the server and returning the response to the client. Commercial ORB's are known in the art and a common type is IBM's System Object Model ("SOM"). On the client side, the ORB handles requests for the implementation of a method and the related selection of servers and methods. When a client application sends a request to the ORB for a method to be performed on an object, the ORB validates the arguments contained in the request against the interface for that object and dispatches the request to the server application, starting the server application if necessary. On the server side, the ORB uses information in the request to determine the best implementation to satisfy the request. This information includes the operation the client is requesting, what type of object the operation is being performed on, and any additional information stored for the request. In addition, the server-side ORB validates each request and its arguments. The ORB is also responsible for transmitting the response back to the client.

Both the client application and the server application must have information about the available interfaces, including the objects and operations that can be performed on those objects. To facilitate the common sharing of interface definitions, OMG proposed the Interface Definition Language ("IDL"). IDL is a definitional language (not a programming language) that is used to describe an object's interface; that is, the characteristics and behavior of a kind of object, including the operations that can be performed on those objects and the parameters to those operations.

IDL is designed to be used in distributed object systems implementing OMG's CORBA Revision 2.0 specification. In a typical system implementing the CORBA specification, interface

definitions are written in an IDL-defined source file (also known as a "translation unit"). The source file is compiled by an IDL compiler that generates programming-language-specific files, including client stub files, server stub files, and header files. Client stub files are language-specific mappings of IDL operation definitions for an object type into procedural routines, one for each operation. When compiled by a language-specific compiler and linked into a client application, the stub routines may be called by the client application, for example, to formulate a request. Similarly, the server stub files are language-specific mappings of IDL operation definitions for an object type (defined by an interface) into procedural routines. When compiled and linked into a server application, the server application can call these routines when a corresponding request arrives. Header files are compiled and linked into client and server applications and are used to define common data types and structures.

In general, computer systems use one of three communication styles: (1) One-way communication; (2) Synchronous Communication; and (3) Asynchronous communication. If a client application in a distributed object system invokes a one-way request, the application sends the request and continues with other work without checking to see if the request was completed. The request truly would go only one way; the application sends the request to the server, but nothing ever returns from the server. If a client application invokes a synchronous communication request, the application transfers control to the ORB and cannot do anything until the request completed or failed. Synchronous communication are most appropriate when an application needed to send and complete requests in a certain order and the operations were of short duration. If an application invokes an asynchronous object call, the application does not wait for the request to complete before it continued with other work. In time-critical situations, asynchronous communication is the preferred style for object calls. Similarly, the implementations of objects in server applications could benefit if asynchronous communication were efficiently supported.

Unfortunately, conventional distributed object systems do not support true asynchronous communication for object calls from a client application. For instance, the CORBA specification offers "deferred synchronous communication." In deferred synchronous communication, a requesting application periodically checks to see if the request has completed by continuously polling using the *CORBA_Request_get_response* operation or the *CORBA_get_next_response*

routine. This process is time-consuming and lacks the benefits of true asynchronous communication.

Moreover, conventional distributed object systems support "threads" which are streams of execution that branch off from a process to handle asynchronous operations. Threaded execution, however, has its own disadvantages such as the inconvenience of having to synchronize access to common data, overall performance loss, and platform dependence.

Accordingly, there is a need for a method and apparatus for performing true asynchronous object calls within a distributed object system.

Moreover, there is a need for a method and apparatus for performing asynchronous object implementations without the use of threaded execution.

SUMMARY OF THE INVENTION

The present invention provides a method and apparatus for performing asynchronous object calls. The present invention also satisfies the need for a method and apparatus for performing non-threaded asynchronous object implementations.

In a preferred embodiment, the method for performing asynchronous object calls of the present invention involves invoking an operation on an object by calling a stub function from a client application. The client application provides the stub function with the input parameters to the operation along with a pointer to a completion routine. The invocation is sent to a server application using an execution environment common to the client and server application. The server application implements the operation on the object and provides a response to the execution environment. Once the operation has been implemented by the server application, the execution environment calls the completion routine with the operation's output parameters. The completion routine should also determine whether or not the object call was successful.

Implementations can be performed asynchronously as well. During an asynchronous implementation, a client application requests that an operation be performed on an object (the request may be made synchronously or asynchronously as stated above). The request is transmitted to a server application by an execution environment accessible by both the client and the server applications. When the request is transmitted, the server application associates the call with a call identifier. If an asynchronous method is called from the server application, the server

application passes the call identifier and the address to a response function to the asynchronous method. The asynchronous operation completes and calls the response function which, in turn, responds to the caller.

By using these "callback" functions ("completion routine" and "response function"), both the client and server applications can continue doing other work without waiting within a particular function. Moreover, by permitting both client and server to execute asynchronously, different invocation styles may be used to suit a particular task. For instance, an object call may be performed asynchronously while its implementation is performed synchronously, and vice-versa, thus providing greater flexibility to the developer.

A more complete understanding of the method and apparatus for asynchronously calling objects will be afforded to those skilled in the art, as well as a realization of additional advantages and objects thereof, by a consideration of the following detailed description of the preferred embodiment. Reference will be made to the appended sheets of drawings which will first be described briefly.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a block diagram of a client/server computing system utilizing the method of the present invention.

Figs. 2a and 2b are diagrams of alternative configurations for Common Execution Environment capsules.

Fig. 3 is a diagram of a Common Execution Environment capsule and its core components.

Fig. 4 is a diagram of the compilation and linking of IDL source code into client and server applications.

Fig. 5 is a flow chart describing the steps involved in a synchronous object call

Fig. 6 is a flow chart describing the steps involved in an asynchronous object call.

Fig. 7 is a section of programming code performing an asynchronous object call.

Fig. 8 is a flow chart describing the steps involved in an asynchronous invocation of an object.

Fig. 9 is a section of programming code performing an asynchronous object

implementation.

Fig. 10 is a flow chart describing a second embodiment of the method of the present invention.

Fig. 11 is a flow chart describing the steps involved in the method of the present invention.

Fig. 12 is a flow chart describing the steps involved in an alternative embodiment of the method of the present invention.

Fig. 13 is a diagram showing a PIF data structure.

Fig. 14 is a diagram showing an entry data structure.

Fig. 15 is a diagram showing an operation data structure.

Fig. 16 is a diagram showing a union data structure.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Reference will now be made in detail to the preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts.

I. System Overview

As illustrated in Figure 1, the method of the present invention is designed for use in a distributed (client/server) computing environment 10. The client and server systems are connected by network connections 12, such as Internet connections or the connections of a local area network. A server computer 11 communicates over a bus or I/O channel 20 with an associated disk storage subsystem 13. The server system 11 includes a CPU 15 and a memory 17 for storing current state information about program execution. A portion of the memory 17 is dedicated to storing the states and variables associated with each function of the program which is currently executing on the server computer. A client computer 21 similarly includes a CPU 27 and associated memory 23, and an input device 29, such as a keyboard, or a mouse 31 and a display device 33, such as a video display terminal ("VDT"). The client CPU 27 communicates over a bus or I/O channel 40 with a disk storage subsystem 33 and via I/O channel 41 with the keyboard 29, VDT 33 and mouse 31.

The client memory 23, preferably, includes a client application 77 that is linked to client stubs 79 (as discussed below) and loaded therein. Similarly, the server memory 17 includes a server application 87 linked to server stubs 89. In addition, both the client memory and the server memory include an execution environment ("CEE") 75, 85 (as discussed below).

The client/server model as shown in Figure 1 is merely demonstrative of a typical client/server system. Within the context of the present invention, the "client" is an application that requests that operations be performed on an object while the "server" is an application that implements the operation on the object. Indeed, both the client and server application may reside on the same computer and within a common capsule, as discussed below. Most likely, however, the client and server application will reside on separate computers using different operating systems. The method of the present invention will be discussed with reference to two capsules running on separate machines.

II. Distributed Computing Environment

The method and apparatus of the present invention may be utilized within any distributed computing environment. In a preferred embodiment, the Common Execution Environment ("CEE") 75, 85, which is a component of the Tandem Message Switching Facility ("MSF") Architecture, is used. The CEE activates and deactivates objects and is used to pass messages between client and server applications loaded in CEE capsules. The CEE may be stored in the memory of a single machine. More likely, however, the CEE and client and server applications will be loaded on multiple machines across a network as shown in Figure 1. The client-side CEE 75 is stored in the client memory 23. The server-side CEE 85 is stored in server memory 17.

The CEE uses a "capsule" infrastructure. A capsule encapsulates memory space and one or more execution streams. A capsule may be implemented differently on different systems depending upon the operating system used by the system. For instance, on certain systems, a capsule may be implemented as a process. On other systems, the capsule may be implemented as a thread. Moreover, client and server applications may be configured within different capsules contained on different machines as shown in Figure 1. Alternatively, the different capsules may be configured as shown in Figure 2. Figure 2a shows a client application 77 loaded in a single capsule 81 and a server application 87 may be loaded in a separate capsule 85. Both capsules,

however, are stored on the same machine 21. Both the client and server applications may also be loaded within a single capsule 81 on the same machine 21 as shown in Figure 2b. As stated above, the method of the present invention will be described with reference to the multiple capsule, multiple machine case. Accordingly, the client 12 and server machine 11 include a client-side CEE 75 and a server-side CEE 85 loaded in their respective memories.

Figure 3 shows a CEE capsule 70 contained, for example, in a client computer memory 27 (not shown) that includes the CEE 75 and certain of the core CEE components and implementations of objects contained within Implementation Libraries 71. The Implementation Libraries 71 include the client application 79 (or the server application in the case of the server capsule) and client stubs 77 (or server stubs) generated from the IDL specification of the object's interface, as described below. The Implementation Libraries 71 and the CEE 75 interact through the down-calling of dynamically-accessible routines supplied by the CEE and the up-calling of routines contained in the Implementation Library. The CEE 75 can also receive object calls 82 from other capsules within the same machine and requests 84 from other CEE's. The client-side CEE 75 and the server-side CEE 85 may communicate using any known networking protocol.

Objects implemented in a CEE capsule may be configured or dynamic. Configured objects have their implementation details stored in a repository (such as the MSF Warehouse 85) or in initialization scripts. Given a request for a specific object reference, the CEE 75 starts the appropriate capsule based on this configuration data. The capsule uses the configuration data to determine which Implementation Library to load and which object initialization routine to call. The object initialization routine then creates the object. Dynamic objects are created and destroyed dynamically within the same capsule. Dynamic objects lack repository-stored or scripted configuration information.

The following paragraphs describe a system-level view of how the Implementation Libraries interact with the CEE 75. The CEE 75 implements requests to activate and deactivate objects within a capsule. In addition, the CEE facilitates inter-capsule object calls 82 as well as requests from other CEE's 84, as discussed above. Object activation requests arise when an object call from a client or server application must be satisfied. To activate an object, the CEE 75 loads the appropriate Implementation Library (if not already loaded) containing the object's methods and then calls a configured object initialization routine contained in the Implementation

Libraries, as discussed below. The initialization routine specifies which interface the Implementation Libraries support and registers the entry points of the object's methods to be called by the CEE at a later time.

When the client and server systems start, both the client-side and server-side CEE's run their own initialization. This initialization tells client and server CEE's where to locate the various Implementation Libraries. Once located by the CEE, the CEE calls the initialization routines in the client and server applications. The initialization routines contained in the client and server applications must first carry out any required application-specific initialization. Next, both the client and server initialization routines call a generated stub function which, in turn, down-calls a CEE function (contained in a dynamic library as stated above) called CEE_INTERFACE_CREATE to specify the object's interface. An interface may be specified for each object. The interface description is normally generated from an IDL description of the interface, as discussed below. CEE_INTERFACE_CREATE creates an interface and returns an "interface handle" to the newly created interface. The handle is a unique identifier that specifies the interface. The server application initialization routine then uses the interface handle to down-call CEE_IMPLEMENTATION_CREATE. CEE_IMPLEMENTATION_CREATE creates an implementation description that can be used by one or more objects. CEE_IMPLEMENTATION_CREATE returns an "implementation handle" that is a unique identifier specifying the implementation for each operation in the interface. Finally, the server application initialization routine uses the implementation handle to call a stub function which down-calls CEE_SET_METHOD. CEE_SET_METHOD specifies the actual addresses of specific method routines of the implementation as contained in the server application. The CEE then has sufficient information to connect object calls in the client application to specific methods in the server application.

III. Compiling and Linking IDL Source Files

Figure 4 shows how IDL source files are compiled and linked into client and server applications that will utilize the method and apparatus of the present invention. First, an IDL source file 101 is prepared containing IDL interface definitions. An IDL compiler 103 compiles the source file 101. The IDL compiler 103 parses the code 101 to produce an intermediate

Pickled IDL file ("PIF") file 105 for storage of the original source file. The generation of a PIF file is described below in Section VIII. A code generator 111 then parses the PIF file. Preferably, however, the IDL compiler and code generator are combined to generate code directly from the source file. The code generator 111 generates files in the language of the client and server applications. If the client and server applications are in different languages, different code generators 111 are used. Alternatively, the code generator 111 and the IDL compiler 103 may be combined in a single application to produce language-specific code. The code generator 111 produces a client stub file 77 containing client stub functions and a server stub file 87 containing definitions for object implementations. The client stub file 77 and the server stub file 87 are compiled by programming language-specific compilers 121, 123 to produce compiled client stub object code and compiled server stub object code. Similarly, a client application 79 and a server application 89 are compiled by programming-language-specific compilers to produce compiled client application object code and compiled server application object code. The client application 79 and the server application 89 also include a header file 119 generated by the code generator 111. The header file 119 contains common definitions and declarations. Finally, a language compiler 121 links the client application object code and the client stub object code to produce an implementation library 71. Similarly, a second language compiler 123 links the server application object code and the server stub object code to produce another implementation library 81.

IV. Asynchronous Client and Server Stub Files

The code generator 111 generates both synchronous and asynchronous client stub functions in the client stub file 77 for each operation in each interface defined in the IDL source file 101. Each stub function corresponds to a particular operation in the interface(s) defined in the IDL source file 101. The client application 79 calls these stub functions to request that an operation be performed on an object. The synchronous client stub functions receive the operation's input parameters, and a reference to the object (in the form of a proxy handle). The synchronous client stub function, in turn, contains a call to CEE_OBJECT_CALL (discussed below) which is used to make an object call through the client-side CEE. The asynchronous client stub functions receive the input parameters to a requested operation, the proxy handle, and

a pointer to a completion routine address. These parameters are, in turn, passed to a CEE function, CEE_OBJECT_CALL_POST, which is also discussed in greater detail below.

The code generator 111 similarly generates synchronous and asynchronous server stub functions in the server stub file 87. Each stub function corresponds to a particular operation in the interface(s) defined in the IDL source file 101. The server application calls these stub functions to notify the CEE of which methods to up-call in order to implement the interface operation that corresponds to the stub. The synchronous server stub function for each operation receives an implementation handle (returned from the call to CEE_IMPLEMENTATION_CREATE) and the address of a method in the server application. The synchronous server stub function, when called in the server application, calls CEE_SET_METHOD to connect the server application method to the operation specified by the stub. When requests for particular operations arrive at the server-side CEE, the CEE up-calls the appropriate method in the server application based upon the address that has been set for the requested operation. When the server application method returns, the server-side CEE will transmit a response back to the client.

The code generator 111 generates a stub function to handle asynchronous object implementation. The asynchronous server stub function is similar to the synchronous stub function, in that the asynchronous stub function receives (from the server application) the implementation handle and the address of a method to up-call from the server application. The asynchronous stub function, in turn, calls CEE_SET_METHOD, which connects the server application method to the operation corresponding to the stub. This call to CEE_SET_METHOD also notifies the CEE (via a parameter to CEE_SET_METHOD) that the method is to be implemented asynchronously. Since the up-called server application method is asynchronous, the CEE will respond to the client upon a call to a generated response stub function also contained in the server stub file 87. The generated response stub function receives the operation's output parameters and a call identifier. These parameters are passed to a CEE function, CEE_RESPOND (described below), which responds to the client application.

V. Synchronous and Asynchronous Calls and Implementations

A. Synchronous Call/Synchronous Implementation

Now, with reference to Figures 5-10, the method of the present invention will be described. This example assumes that a server object to be called was previously activated by the server-side CEE. As stated above, object calls may be performed synchronously or asynchronously through generated synchronous and asynchronous client stubs. Similarly, object implementations in the server application may be performed synchronously or asynchronously. The method and apparatus described herein involves separate capsules loaded on separate machines.

First, a synchronous object call combined with a synchronous object implementation will be described. Figure 5 shows a flow chart describing the steps involved in a synchronous call to an object and a synchronous implementation of the object. In a first step 501, an object reference for the desired object must be obtained by the client application. The object reference may be obtained in a number of ways. Client applications usually receive the reference from configuration data, directories or invocations on other objects to which they have object references. An object reference can be converted to a string name that can be stored in files to be accessed later.

The object call is initiated in step 503 by first obtaining a "proxy handle" to the object reference from the CEE. The proxy handle is a structure containing the object reference along with information regarding calls made to the same object. The proxy handle is designed to facilitate multiple calls to the same object without incurring the overhead that can occur in calling the object each time. By creating a proxy handle via which object calls can be made, certain initialization routines may be performed once through the proxy handle while allowing multiple simultaneous calls to the proxied object. A proxy handle is created in CEE 75 by down-calling the CEE function CEE_PROXY_CREATE from the client application. That function is defined in the C programming language as follows:

```
CEE_PROXY_CREATE (
        const char      *objref,
        const char      *intf_handle,
        char            *proxy_handle);
```

The function receives the object reference, *objref*, and an interface handle, *intf_handle* returned

by CEE_INTERFACE_CREATE (described above), and returns a proxy handle identifying the newly created proxy object. The details of proxy creation and a description of the proxy handle structure returned by CEE_PROXY_CREATE are described below in Sections VI and VII.

The proxy handle returned by CEE_PROXY_CREATE is a structure containing a pointer to the object if the object is in the same capsule or a pointer to a client port if the object is not in the same capsule. The client-side CEE also uses the proxy to store a completion routine address that is used in an asynchronous object call (discussed below). The completion routine will be discussed below with reference to asynchronous calls from the client application. In a synchronous object call, the address of a completion routine is not stored in the proxy handle structure.

In step 505, the client calls the object. Specifically, the client application requests that an operation of an object's interface be performed on the object by calling the appropriate generated synchronous stub function which, in turn, contains a down-call to CEE_OBJECT_CALL. CEE_OBJECT_CALL is defined in C as follows:

```
CEE_OBJECT_CALL (
        const char      *proxy_handle,
        long            *operation_idx,
        void            *param_vector);
```

The stub function receives the proxy handle and the input parameters to the operation. The stub function provides CEE_OBJECT_CALL with three parameters. The first parameter is the *proxy_handle*. This parameter specifies the object to be called and will be used to respond to the call. The second parameter, *operation_idx*, is an integer that specifies which of the object's methods is to be called. The identifier is used locally by the client to specify a particular operation. The identifier saves the client the trouble of repeatedly performing string comparisons on the operation name. Similarly, on the server side, the server specifies operations using its own operation identifier. The *param_vector* is an array containing the addresses of the object call's input parameters, output parameters and exception structure. The address of the exception structure preferably is the first element in the array. If the operation is not of type *void*, then the following element in the array contains the address of the variable to receive the operation's

result. The remaining elements contain the address of the operation's input and output parameters in the same order as they were defined in IDL.

In step 507, the request for the operation to be performed, including the input parameters, operation identifier, and object reference (as determined by the proxy handle), is transmitted from the client-side CEE to the server-side CEE via the CEE interface 84. As stated above, the CEE's may utilize any method of data transfer to transfer the call across systems or networks.

To implement the object synchronously, the server-side CEE 85 calls the appropriate method in the server application (as specified by the initialization routine's call to a server stub function which, in turn, called CEE_SET_METHOD). Preferably, the server-side CEE 85 provides three parameters to the method in the server application: (1) An optional call identifier that is used by the server application method to allocate and deallocate memory for the call; (2) An exception identifier to hold error information that can be returned to the caller; and (3) The input parameters to the requested operation in the order as they were originally defined in IDL. The method uses the input parameters and carries out the request in step 509. The call identifier can be used to call other CEE functions which automatically allocate and deallocate memory within the server application method, as described below in connection with Figure 10.

If the operation requested by the client includes an output parameter, the method in the server application will initialize and modify this output parameter (if necessary) within the body of the method. Once the method completes, it returns the output parameter to the server-side CEE in step 511. The server-side CEE automatically sends the response to the calling function in the client application in step 513. The response message contains the output parameters (if any) along with optional server-side-CEE-generated exception information regarding the success or failure of the operation. The object call is then complete.

B. Asynchronous Call/Synchronous Implementation

Figure 6 shows the steps necessary to perform an asynchronous object call from the client application combined with a synchronous implementation. The first few steps are similar to the steps involved in the synchronous case. In step 601, the client application obtains an object reference. In step 603, the client application creates a proxy handle for the object reference using CEE_PROXY_CREATE.

Next, in step 605, the object is called using a generated asynchronous client stub function. This generated asynchronous stub function receives the object's proxy handle, the input parameters to the operation, a call tag (described below) and the address to a "completion routine" within the client application. The completion routine will be called by the client-side CEE when a response to the object call has returned. The asynchronous stub function, in turn, down-calls a client-side CEE function, CEE_OBJECT_CALL_POST, which calls the object. CEE_OBJECT_CALL_POST is defined in C as follows:

```
CEE_OBJECT_CALL_POST (
        const char      *proxy handle,
        long            operation_idx,
        const void      *const *param_vector,
        void            completion_routine,
        char            call_tag1);
```

The stub function provides the proxy handle for the requested object and an operation index that specifies which of the object's operations is to be performed. The *param_vector* parameter supplied here is an array containing the addresses of the object call's input parameters only. The input parameters are stored in the array in the same order as they were defined in IDL to permit object calls across multiple platforms. The *call_tag1* parameter is a constant used to identify this call to the object

The asynchronous client stub function and CEE_OBJECT_CALL_POST also receive the address of a "completion routine" in the client application that will be called by the client-side CEE 75 when a response to the object is returned to the client application or an error condition (or other exception) has been received by the client-side CEE. In step 607, the client-side CEE stores the completion routine address in the proxy handle for later use. When the call returns for a particular proxy handle, the client-side CEE will extract the completion routine address from the proxy structure and call the completion routine. The completion routine will be discussed below. If multiple calls are made requiring the same completion routine, the *call_tag1* parameter may be used to identify a particular call within the completion routine. The *call_tag1* parameter is also stored in the proxy structure in step 607.

In step 609, the request for an operation to be performed on an object is transmitted from the client-side CEE to the server-side CEE using any transport mechanism. Once the object call has been made, the client can continue performing other functions. It need not wait for a response as in the synchronous case. Moreover, the client application is not required to continuously poll the server for a response.

On the server side, the object is implemented synchronously as described above. The server-side CEE selects the appropriate method in the server application to perform the operation. The method implements the operation on the requested object in step 611 and responds to the client application in step 613. The server-side CEE transmits the response containing any output parameters and exception information to the client-side CEE in step 615.

The client-side CEE locates the proxy handle structure for the transmitted response in the client CEE capsule. In step 617, the client-side CEE extracts the completion routine address and *call_tag1* identifier from the proxy structure. The client-side CEE calls the completion routine in the client application, in step 619. The client-side CEE provides the completion routine with the exception information, the output parameters of the object's methods and an optional identifier tag (as specified above by *call_tag1*) to identify which asynchronous call has completed (if multiple calls use the same completion routine). The completion routine in the client application can then use these parameters as necessary.

Figure 7 shows sample code in a client application 77 that uses the method of the present invention. The client application is written in the C programming language. In this example, the client obtains the current time from an object of the Time interface (the "Time object"). The client obtains the time by requesting the performance of the Tim_Now operation on the Time object. The Tim_Now operation takes a constant (LOCAL or GMT) as an input parameter and returns the local time or Greenwich Mean Time.

At line 701, the client application includes the header file 119 generated by the code generator from an IDL source file containing the Time interface definition. The header file 119 includes the synchronous and asynchronous client (and server) stub functions for each operation in the Time interface, including the Tim_Now stub functions.

The client initialization routine is up-called by the client-side CEE 75 at line 702. The client registers the Time interface with the client-side CEE 75 at line 703 by calling an interface

stub function which will call CEE_INTERFACE_CREATE. The CEE returns an interface handle, *intf*, to the client. (The variable *sts* (status) is a dummy variable that allows the function return value to be examined when debugging.)

At line 703, the client obtains an object reference from a configuration file previously registered with the CEE. The reference, *objref*, refers to an object of the Time interface. Because the client application plans to call TIM_Now twice and simultaneously, the client uses the object reference to create two proxy handles at lines 705 and 706. The client calls CEE_PROXY_CREATE which returns *proxy* and *proxy1*.

At line 707, the client calls the Time object using the asynchronous stub function, Tim_Now_pst. The client provides the first proxy handle (*proxy*), a completion routine tag (*LOCAL_TIME_TAG*), and an input parameter (*LOCAL*). At line 708, the client continues to perform other asynchronous work by making another call to the same object using a different proxy handle (*proxy1*), completion routine tag (*GMT_TAG*), and input parameter (*GMT*). If the first call were made synchronously, the client application would be required to wait until the first call to Tim_Now returned before making the second call to Tim_Now.

For both calls, the client-side CEE stores the completion routine address and completion routine tag in the proxy structure. The client-side CEE 75 transmits both requests to the server-side CEE 85. The server application contains an implementation for the Tim_Now operation. This implementation (not shown) provides the local or GMT time. The time is inserted into an output parameter and the output parameter and any exception information are transmitted back to the client-side CEE 75.

The client-side CEE determines which call has returned and calls the completion routine for the returning call. The completion routine of the example is shown beginning at line 709. A switch statement at line 710 receives the optional completion routine tag parameter. If the tag value is *LOCAL*, the function will print the local time as shown at line 711-712. If the tag value is *GMT*, the function prints the GMT time as shown at line 713-714. Another example of a completion routine tag would be that the CEE assigns sequential tag numbers each time it makes a new function call to an object.

B. Asynchronous Implementation

Next, with reference to Figure 8, the asynchronous implementation of an object by the server application will be described. Asynchronous implementation involves an original method in the server application calling a second asynchronous method. The original method may be called by the client synchronously or asynchronously. The method described herein allows an object to support more than one concurrent request. In a first step 801, the client calls the object and requests that an operation be performed on the object. The object call, as discussed above, may, itself, be synchronous or asynchronous. If the call is synchronous and the implementation of the object is asynchronous, however, the client and the object must be running in different capsules in an MSF system. This discussion assumes that the client application and server applications are running in separate capsules on different machines. In step 803, the call is transmitted from the client-side CEE to the server-side CEE via interface 84 using any known transport mechanism.

On the server side, the CEE calls the appropriate server application method for the requested operation in step 805. This method (hereinafter called the "original method") was specified as the appropriate method for the requested operation by a call to an asynchronous stub function from the initialization routine, as discussed above. The CEE provides the original method in the server application with the same parameters as in the synchronous case: (1) A *call_id* parameter; (2) An *exception* parameter used to track errors in implementing the object; and (3) The input parameters to the operation. In the asynchronous case, however, the call identifier provided to the server application method is stored by the server-side CEE in the server computer memory 17. Since the server-side CEE will respond to the client-side CEE upon a call to CEE_RESPOND (rather than automatically upon a return from the original method), the server-side CEE will track each call with a different call identifier. Accordingly, the original method, any asynchronous methods that are called from the original method, and the response function (discussed below) must also keep track of the call identifier. Within the original method, in step 807 a *context* variable is preferably used to store the *call identifier* passed to the original method. The context variable is also used to store the output parameters containing the result of any operations performed by the method. The *context* variable will be passed to the asynchronous method called from the original method. Each time, the CEE up-calls the original method, a new *context* variable is created.

In step 809, the original method calls an asynchronous method to carry out another function. For example, the original method may need to perform an asynchronous input/output operation such as opening or closing a disk file. Alternatively, the original method may make an asynchronous object call, itself, to carry out some function. If the original method calls an asynchronous method, the original method preferably provides the asynchronous method with the address of a response function in the server application. When the asynchronous method completes, the asynchronous method will call the response function. In addition, the asynchronous method receives the *context* variable containing the *call identifier* and the result of any operations performed in the original method. The *context* parameter will ultimately be used by the response function in order to associate the response to a particular object call.

The asynchronous method performs its designated function. When completed, the asynchronous method calls a response function in step 811. The asynchronous method passes the *context* parameter containing the *call identifier* and output parameters (as well as any other context that may be useful) to the response function. The response function, in turn, calls the asynchronous response stub function in the server application in step 813. As discussed above, the response stub function contains a down-call to CEE_RESPOND. If the original method had not called an asynchronous method, but was specified as an asynchronous method in the initialization routine, the original method could have called the asynchronous response stub function directly. Alternatively, any method in the server application can call CEE_RESPOND.

CEE_RESPOND is defined in C as follows:

```
CEE_RESPOND (
        const char      *call_id,
        const void      *const *param_vector);
```

The response function transmits the call identifier to the stub function by extracting the call identifier from the context. The stub function transmits the identifier to CEE_RESPOND. The server-side CEE locates the call identifier in the server computer memory and responds to the appropriate call based upon the call identifier. The response contains the *param_vector* parameter which is an array containing pointers to the object call's output parameters and exception structure. The first element of the array is the address of the exception structure. If

the operation is not of type *void*, then the next element contains the address of the variable containing the operation's result. Subsequent elements contain the addresses of the operation's output parameters in the same order as they were defined in IDL.

In step 815, the response is sent back to the client-side CEE via interface 84. The client-side CEE up-calls the appropriate method in the client application and provides the output parameters and exception information from the call to CEE_RESPOND.

Figure 9 shows sample code in a C-language server application that implements objects asynchronously. This example contains a possible implementation for the object call shown in Figure 7. This code contains an asynchronous implementation of a Tim_Now operation on an object of the Time interface. (Data type definitions have been omitted).

The server-side CEE 85 calls the server application's initialization routine when the implementation libraries are loaded. The routine, beginning at line 901, obtains an interface handle by calling a server stub function at line 902. The interface handle is used to create an implementation handle at line 903 which, in turn, is passed to a server stub for setting the address for a method in the server application. At line 904, the address for the server method, NowMethod, is specified as the method to be up-called by the server-side CEE for the Tim_Now operation. Moreover, by calling an asynchronous stub function to set the address, server application notifies the server-side CEE that a call to CEE_RESPOND is required before responding to the client (rather than responding automatically upon exiting the up-called method). Further, the server-side CEE 85 will pass a call identifier into the method to identify the call.

When a call for TIM_Now arrives at the server-side CEE, the CEE up-calls NowMethod(the original method) in the server application. At line 905, NowMethod converts the time to the requested time zone and stores it in a variable, *timeptr*. At lines 906-907, the method stores the result in a predefined context structure. At line 908, this structure is also used to store the call identifier passed into the method at line 920.

At line 909, the original method (NowMethod) calls an asynchronous method. In this example, the asynchronous method is a trivial down-call to a CEE timer function. The CEE timer method receives the context structure and the address of a response function, TimerExpired, in the server application. The CEE will call the response function when the timer "pops" after one second. Execution on the server side continues after the call of line 909 without

waiting for control to return from CEE_TIMER_CREATE.

When the timer pops, the CEE calls the response function, TimerExpired. TimerExpired, at line 910, contains a single call to a server stub response function. The server stub response function, preferably, calls CEE_RESPOND. The call to the server stub response function extracts the call identifier and the output parameter, *timestr*. The server-side CEE transmits the output parameter (and exception information, if any) back to the client-side CEE 75, in a manner dependent on whether the original call was made synchronously or asynchronously.

C. Asynchronous Implementation With Memory Allocation

Because multiple calls can be made to the same method in the server application, the method should preferably have some method for allocating and deallocating memory for numerous calls. In an alternative embodiment of the method and apparatus of the present invention, memory allocation and deallocation ("garbage collection") during object implementation is provided. Figure 10 shows this alternative method of the present invention. Steps 1001-1005 are similar to the steps involved in Figure 6. Thus, in step 1001, the object is called asynchronously or synchronously. Next, in step 1003, the request is transported from the client-side CEE to the server-side CEE. The server-side CEE then up-calls the appropriate method in the server application in step 1005.

In step 1007, the server application allocates memory for the call. This is accomplished by calling an appropriate memory allocation function, such as MALLOC in C and C++. A mechanism must also be provided to deallocate previously allocated resources. Preferably, memory allocation and deallocation is performed by down-calling CEE_TMP_ALLOCATE, defined in C as follows:

```
CEE_TMP_ALLOCATE (
        const char      *call_id
        long            len
        void            **ptr
```

The *call_id* parameter identifies the particular call so that the CEE can automatically deallocate

the memory upon completion of the call. The *len* parameter specifies the number of bytes to allocate. The CEE returns the address of the allocated memory using the *ptr* parameter.

Next, the method is performed and the result and *call_id* are stored in the *context* variable in step 1009. The method then calls an asynchronous method (or object call) in step 1011. The asynchronous operation performs its functions and then calls the response function in step 1013. The response function, in turn, calls a server stub response function in step 1015. The output parameters to the operation along with any exception information indicating the success or failure of the object call are sent back to the caller in step 1017.

VI. Proxy Creation and Deletion

Now, with reference to Figures 11 and 12, the method of the present invention will be described. The method of the present invention will be described with reference to a client application executing in a capsule on a client computer and a server application executing in a separate capsule on a server computer. Figure 11 is a flow chart depicting the steps involved in utilizing a proxy handle to call an object. In a first step 1101, an object reference for the desired object must be obtained. The object reference may be obtained in a number of ways. Client applications usually receive the reference from configuration data, directories or invocations on other objects to which they have object references. An object reference can be converted to a string name that can be stored in files to be accessed later.

Once the object reference has been obtained, the object call may be performed. In the method of the present invention, the object call is performed by first obtaining a "proxy handle" to the object reference. The proxy handle is a unique identifying data structure (a "proxy object") for a particular object reference. The proxy structure contains information about an object and calls to that object. Calls can be made to the specified object using the proxy handle. The proxy handle facilitates calls to the same object and prevents overhead that occurs in multiple calls to the same object. By creating a proxy handle via which object calls can be made, certain initialization routines may be performed once through the proxy handle while allowing multiple calls to the proxied object. In addition, the proxy handle facilitates the use of asynchronous calls to an object (discussed below). In a preferred embodiment, a proxy handle is created in step 1103 by down-calling the client-side CEE function, CEE_PROXY_CREATE, in the client

application. That function is defined in the C programming language as follows:

```
CEE_PROXY_CREATE (
        const char      *objref,
        const char      *intf_handle,
        char            *proxy_handle);
```

The function receives the object reference, *objref*, and an interface handle, *intf_handle*, and returns a proxy handle identifying the newly created proxy object. As discussed above, in connection with Figure 3, an interface must be created for each object. An interface defines the operations available for a collection of similar objects. The interface is defined in IDL and compiled and linked into a client application. The client application calls CEE_INTERFACE_CREATE in its initialization routine to specify the interface. The client-side CEE returns an interface handle that can be used to create any number of objects or proxies.

In a preferred embodiment of the present invention, the proxy object is represented by a structure containing the following fields:

```
link;
call_link;
self;
nor;
state;
call_active;
destroy;
lock_count;
*intf;
call_compl_rtn;
call_compl_tag1;
call_complt_tag2;
call_compl_sts;
operation_idx;
*client_allocated_params;
```

**server_allocated_params*;
**obj*;
operation_idx_table;
max_response_size;
**req_area*;
**rsp_area*;
**rsp_param_buf*;
ochan;

Each of the components of the proxy object data structure will now be discussed. The addresses and values stored in each of these components is modified by the client-side CEE with each call to the object referred to by the proxy structure. The client-side CEE maintains a linked list of proxy structures. The *link* member of each proxy structure contains a pointer to the next entry in this list of proxy structures.

In a preferred embodiment, object calls can be either synchronous (client application requests that an operation be performed on an object and waits for a response) or asynchronous (client application requests that an operation be performed on an object and continues to do other work). When asynchronous object calls are dispatched in the same capsule, each call is queued onto a linked list contained in an object structure that exists for every object when activated. The *call_link* parameter is a link to the list of calls. The *self* member is the handle of this particular proxy structure. This handle is returned to the client during CEE_PROXY_CREATE.

The object reference passed to CEE_PROXY_CREATE is stored in the *nor* member. The *state* parameter indicates whether the proxy structure includes a pointer to an internal object structure, an external object (via a client port), or a non-existing object structure (is stale). If the proxy is internal, a pointer to the object is contained in the *obj* parameter. If the object is in a different capsule, the *ochan* parameter contains a pointer to a client port handle or other information required to communicate with the object.

The *call_active* member holds a true/false value. The *call_active* member is set to true if an object call is outstanding for this particular proxy handle. Only one object call can be outstanding on a given proxy. The *lock_count* member is incremented to prevent the proxy

structure from being destroyed. It is decremented when the structure is no longer needed. The *destroy* member is a true/false value that is set to true if this proxy structure should be destroyed when *lock_count* drops to zero. The *intf* member is the address of the *intf* structure that describes the interface (discussed above).

The next four structure members, *call_compl_rtn*, *call_compl_tag1*, *call_compl_tag2*, *call_compl_sts*, are used to implement asynchronous object calls. Asynchronous calls to an object in a server application are made by passing the address of a completion routine to the client stub function when called. The client stub function, in turn, calls the client-side CEE and provides the completion function address. The client-side CEE stores the completion function address in the *proxy* structure upon creation of the proxy handle. When the object call completes, the client-side CEE calls the completion routine specified in the *proxy* handle. The routine is called to notify the client application that the call has completed. While the object is being implemented, the client application can continue performing other functions. The member *call_compl_rtn* contains the address of the completion routine. Since multiple calls may be made to the same proxied object, the client application can identify the call by using the *call_compl_tag1* parameter when the object call is made. The *call_compl_tag1* identifier is passed to the client stub function. These identifiers are specified in the proxy structure by the members *call_compl_tag1* and *call_compl_tag2*. The *call_compl_sts* indicates the call completion status for asynchronous calls that could not be called.

The *operation_idx* member specifies which of the object's operations is to be called. Operation identifiers are generated by the code generator for each operation in the interface. The *allocated_params* member is a pointer to the parent of temporarily allocated parameters (used for unbounded types and the like). The deallocation of this member performs garbage collection on the next call to the object referenced by this proxy structure. The *operation_idx_table* parameter is a pointer to an operation index translation table that is used only if the object is contained in the same capsule.

Memory allocation is performed utilizing the *max_response_size*, *req_area*, *rsp_area*, and *rsp_param_buf* members. The *rsp_param_buf* member points to a buffer containing the response parameters. The next time that this proxy object is used, the buffer will be deallocated. The *max_response_size* member is the maximum expected response size. This is used to

allocate the *rsp_area* member. The *req_area* member points to an *area* structure that will be used for the request. The *rsp_area* points to an *area* structure that will be used for the response to the object call. The *area* structure contains the following fields, as defined in C:

desc;
**data*;
curlen;

The *area* structure contains an object call area descriptor and a pointer to data and the current length of that data.

The call by the client application to CEE_PROXY_CREATE causes the client-side CEE to automatically allocate memory for the object call in step 1108. Memory in the client computer is allocated along with any additional resources necessary for making the call. Once the proxy handle is destroyed (through a down-call to CEE_PROXY_DESTROY, discussed below), the memory and any allocated resources are freed. Memory allocated for variable-sized output parameters from an object call are deallocated when the next object call is made using the same proxy handle.

The proxy handle is used to make all subsequent calls to the object referred to by the proxy. The object call is made in step 1111 by calling the appropriate stub function in the client application and passing the proxy handle and input and output parameters along with exception information to the function. The stub function, in turn, down-calls CEE_OBJECT_CALL, defined in C as follows:

```
CEE_OBJECT_CALL (
        const char      *proxy_handle,
        long            operation_idx,
        void            **param_vector);
```

The proxy handle is specified by the *proxy handle* parameter. The parameter *operation_idx* specifies which of the object's methods is to be called. This parameter is an index to the required method in the interface description that was supplied when the interface was created. Finally, the *param_vector* parameter is an array of pointers to the object call's input parameters, output

parameters, and exception structure. The address of the exception structure is the first element in the array. If the operation is not of type *void*, then the following element contains the address of the variable to receive the operation's result. Subsequent elements contain the addresses of the operation's input and output parameters in the same order as they were defined in IDL.

The call is then transported to the server using any transport mechanism. In step 1115, the server application implements the object. This is performed by the server-side CEE which up-calls the appropriate method routine. The method routine is passed the *param_vector* parameter containing the addresses of all the input and output parameters. When the method exits, a response is sent to the caller in step 1119 and the object call is complete.

Once the first call has completed, the proxy handle may be used again to make further calls to the same object. Each subsequent call to the object may be made without validating the object or performing other start-up operations. Thus, the proxy creation step can be placed in a non-time-critical portion of the client application and object calls can be made in a time-critical portion of the application.

Following the final object call for a specified proxy handle, the proxy handle is destroyed in step 1121. This is accomplished by calling CEE_PROXY_DESTROY in the client application, defined in C as follows:

```
CEE_PROXY_DESTROY (
        const char          *proxy_handle);
```

The proxy handle is passed to the function. The client-side CEE destroys the proxy handle and frees all previously-allocated resources for the proxy handle in step 1128. Alternatively, an object call may be canceled and all of the resources associated with the call may be deallocated by destroying the proxy while the call is outstanding.

VII. Proxy Creation And Memory Allocation

Figure 12 shows an alternative embodiment of the object call method of the present invention. In this embodiment, memory is allocated during the implementation of the object as well as during the object call.

Steps 1201-1211 are similar to the steps described above. Thus, in step 1201, an object

reference is obtained. Next, a proxy handle is obtained by down-calling CEE_PROXY_CREATE which returns the proxy handle. The object call is then made in step 1211 using CEE_OBJECT_CALL, which is passed the proxy handle of the referenced object.

In step 1213, the server-side CEE up-calls the appropriate method routine in the server application. The method routine, in step 1216 when called, down-calls a server-side CEE function to allocate memory. That function, CEE_TMP_ALLOCATE is defined in C as follows:

```
CEE_TMP_ALLOCATE (
        const char      *call_id,
        long            len,
        void            **ptr);
```

The function uses the *call_id* parameter to track a particular object call. Each call to the object is given a unique *call_id* by the server application. Thus, once the call is made, the server implementation provides an id for the call in the *call_id* parameter. The number of bytes to allocate is specified in the *len* parameter. The function returns the address of the allocated memory through the *ptr* parameter.

The object's method is performed by the server application in step 1219. The server application responds to the caller in step 1215. Upon exiting the method function, the memory allocated under the down-call to CEE_TMP_ALLOCATE is freed in step 1220. The client application then makes another object call in step 1211 or destroys the proxy handle in step 1225. If the proxy handle is destroyed, the memory allocated in step 1209 is automatically deallocated by the client-side CEE in step 1228.

Memory can be prematurely deallocated using CEE_TMP_DEALLOCATE. That function is defined as:

```
CEE_TMP_DEALLOCATE (
        In      void    *ptr);
```

The function is passed the *ptr* parameter that was provided by CEE_TMP_ALLOCATE. The CEE frees the address pointed to by that parameter.

VIII. Creating a Pickled IDL Format Data Structure

The Pickled IDL Format ("PIF") data structure is designed to be used in conjunction with IDL compilers and code generators loaded in the client memory 23 and server memory 17. The data structure is based upon an IDL source file stored in memory 23 or in memory 17. The source file may also be contained on a computer-readable medium, such as a disk. The data structure of the present structure contains a parse tree representing the IDL source file. The data structure can be stored in memory 23 or in memory 17 or on a computer-readable medium, such as a disk. The data structure that represents the source file is referred to as a Pickled IDL Format ("PIF"). The PIF file can be accessed at run-time by clients and servers that use the interfaces defined in the source file. The parse tree contained in the PIF file is an array using array indices rather than pointers. The use of array indices permits the resulting parse tree to be language-independent. The first element of the array is unused. The second element of the array (index 1) is the root of the parse tree that acts as an entry point to the rest of the parse tree.

The data structure, *tu* 1301, is shown in Figure 13, and defined in IDL as follows:

```
struct tu_def {
        sequence<entry_def>     entry;
        sequence<string>        source;
}
```

The data structure 1301 contains a sequence (a variable-sized array) of parse tree nodes 1305, each of type entry_def (defined below) and a sequence of source file lines 1307. The sequence of source file lines 1307 is a sequence of strings containing the actual source code lines from the IDL source file.

Each parse tree node (or "entry") 1305 consists of a fixed part containing the name of the node and its properties as well as a variable portion that depends upon the node's type. The parse tree node is shown in Figure 14 and defined in IDL as follows:

```
struct entry_def {
        unsigned long entry_index;
        string        name;
        string        file_name;
```

```
unsigned long line_nr;
boolean in_main_file;
union u_tag switch (entry_type_def) {
    case entry_argument: argument_def argument_entry;
    case entry_array: array_def array_entry;
    case entry_attr: attr_def attr_entry;
    case entry_const: const_def const_entry;
    case entry_enum: enum_def enum_entry;
    case entry_enum_val: enum_val_def enum_val_entry;
    case entry_except: except_def except_def_entry;
    case entry_field: field_def field_def entry;
    case entry_interface: interface_def interface_entry;
    case entry_interface_fwd: interface_fwd_def interface_fwd_entry;
    case entry_module: module_def module_entry;
    case entry_op: op_def op_entry;
    case entry_pre_defined: pre_defined_def pre_defined_entry;
    case entry_sequence: sequence_def sequence_entry;
    case entry_string: string_def string_entry;
    case entry_struct: struct_def struct_entry;
    case entry_typedef: typedef_def typedef_entry;
    case entry_union: union_def union_entry;
    case entry_union_branch: union_branch_def union_branch_entry;
} u;
};
```

The fixed part of the parse tree node includes *entry_index* 1405, an unsigned long which is the index for this particular entry in the parse tree. The unqualified name of the entry is contained in the field *name* 1407. The name of the original IDL source file is contained in the field *file_name* 1411. The field line_nr 1413 contains the line number in the IDL source file that caused this parse tree node to be created. The boolean *in_main_file* 1415 indicates whether or

not the entry is made in the IDL source file specified on the command line or whether the entry is part of an "include" file. Following these fields, the parse tree node includes a variable portion--a union 1417 having a discriminator, *entry_type_def*. The union discriminator, *entry_type_def*, specifies the type of node and which variant within *entry_def* is active. *Entry_type_def* is an enumeration defined as follows:

```
enum entry_type_def {
        entry_unused,
        entry_module,
        entry_interface,
        entry_interface_Fwd,
        entry_const,
        entry_except,
        entry_attr,
        entry_op,
        entry_argument,
        entry_union,
        entry_union_branch,
        entry_struct,
        entry_field,
        entry_enum,
        entry_enum_val,
        entry_string,
        entry_array,
        entry_sequence,
        entry_typedef,
        entry_pre_defined
};
```

Entry_type_def includes a list of the various types of parse tree entries. Each parse tree entry

represents a constant integer that is used in the switch statement contained in *entry_def*. For each entry, the union *u_tag* will include a different type of structure. The first enumerated value entry_unused corresponds to the value zero and is not used in determining the type of the union.

If the parse tree entry is a module (specified by the value entry_module) the variable portion of the parse tree entry is a data structure including a sequence of module definitions. Each module definition is an unsigned long acting as an index in the parse tree array.

If the parse tree entry is an interface, as specified by the value entry_interface, the variable portion of the parse tree is a data structure including a sequence of local definitions and a sequence of base interfaces from which this interface inherits. If the parse tree entry is a forward declaration of an interface (entry_interface_fwd), the union is an unsigned long containing the index of the full definition.

Constants (entry_const) are represented in a parse tree node as a structure containing the value of the constant. A union and switch/case statement are preferably used to discriminate between the various base type constants (boolean constant, char constant, double constant, etc...) that may be included in the source file.

Exceptions (entry_except) are represented in a parse tree node as a structure containing a sequence of fields. An attributes (entry_attr) is represented as a data structure containing a boolean value that indicates whether the attribute is read-only and an unsigned long that indicates the data type.

If the parse tree entry is an operation (op_def), the variable portion 1417 of the entry data structure 1305 is a data structure as shown in Figure 15. The data structure 1417 contains a boolean 1505 that indicates whether or not the operation has a one-way attribute, an unsigned long 1307 that indicates the return type, a sequence of arguments 1509 to the operation, a sequence of exceptions 1511 to the operation, and a sequence of strings 1513 that specify any context included in the operation. If the parse tree entry is an argument to a particular operation (entry_argument), the variable portion of the parse tree entry is a structure containing unsigned longs that indicate the data type and direction of the argument.

If the parse tree entry is a union (entry_union), it is represented in the parse tree entry as shown in Figure 16. The data structure 1417 contains an unsigned long specifying the discriminator 1603 and an unsigned long specifying the type 1605. The type is preferably

specified using an enumerated list of base types. The structure 1417 further includes a sequence of the union's fields 1607. If the parse tree entry is a union branch (entry_branch), the variable portion of the parse tree entry is a structure containing an unsigned long indicating the base type of the branch, a boolean indicating whether or not the branch includes a case label, and the value of the discriminator. Since the value is of a particular data type, preferably an enumerated list of the various base types is used to specify the value within the structure used to represent the union branch.

For data structures (entry_struct), the variable portion of the parse tree entry includes a structure containing a sequence of the specified structure's fields. Enumerated values (entry_enum) are represented by a structure containing a sequence of enumerated values. Enumerations of an enumerated type (entry_enum_val) are represented in the parse tree entry by a structure containing an unsigned long holding the enumeration's numerical value.

If the parse tree entry is a string (entry_string), the variable portion of the parse tree entry is a structure containing the string's maximum size. A maximum size of zero implies an unbounded string. An array (entry_array) is represented in the parse tree entry by a structure containing an unsigned long holding the array's base type and a sequence of longs holding the array's dimensions. A sequence (entry_sequence) is represented by a structure containing unsigned longs holding the sequence's base type and the sequence's maximum size.

For type definitions (entry_typedef), the parse tree entry includes a structure containing an unsigned long value indicating the type definition's base type. Predefined types (entry_pre_defined) are represented by a structure containing the data type. To specify the type, preferably an enumeration of the various base types are used.

Once the IDL source file has been described using the *tu* data structure, the data structure may be transported to a file or database using any known methods.

Having thus described a preferred embodiment of a method for asynchronously calling and invoking objects, it should be apparent to those skilled in the art that certain advantages of the within system have been achieved. It should also be appreciated that various modifications, adaptations, and alternative embodiments thereof may be made within the scope and spirit of the

present invention. For example, a method has been illustrated, but it should be apparent that the inventive concepts described above would be equally applicable to a non-MSF environment. The invention is further defined by the following claims.

CLAIMS

What is Claimed is:

1. A method for asynchronously performing an operation on an object, the operation being requested by a client application to a server application, the method comprising the steps of:

obtaining an object reference to the object in the client application;

requesting the operation with a stub function in the client application, the stub function being passed the object reference, an input parameter of the operation, and a computer memory address of a completion routine in the client application;

storing the completion routine memory address;

transmitting the input parameter to a method in the server application via an execution environment accessible to the client application and the server application;;

implementing the operation on the object in the server application, the implementation including a response to the client application;

transmitting the response to the client application via the execution environment;

calling the completion routine in the client application, the completion routine being passed the response.

2. The method for performing an operation on an object, as recited in Claim 1, wherein the object reference is obtained from a configuration file accessible to the client application.

3. The method for performing an operation on an object, as recited in Claim 1, wherein the object reference is obtained from a disk file accessible to the client application.

4. The method for performing an operation on an object, as recited in Claim 1, wherein the object reference is obtained from a previous object call by the client application.

5. The method for performing an operation on an object, as recited in Claim 1, wherein the step of implementing the operation on the object is performed asynchronously.

6. The method for performing an operation on an object, as recited in Claim 5, wherein the step of asynchronously implementing the operation on the object further comprises the steps of:

calling an asynchronous function from within the server application method, the asynchronous function being passed a call identifier and a memory address containing a response function in the server application;

calling the response function from the asynchronous function, passing the call identifier to the response function; and

responding to the object call based upon the identifier.

7. A method for asynchronously performing an operation on an object via a request by a client computer application to a server computer application, the method comprising the steps of:

obtaining an object reference to represent the object;

calling the object with a stub function in the client application, the stub function being passed the object reference, an input parameter of the operation, and a client computer memory address of a completion routine in the client application;

transmitting the input parameter and object reference to a method in the server application via an execution environment accessible to the client application and the server application;

calling an asynchronous function from within the method, passing a call identifier and a server computer response function memory address to the asynchronous function;

calling the response function from the asynchronous function, passing the call identifier to the response function;

responding to the object call based upon the identifier;

transmitting the response to the caller via the execution environment;

calling the completion routine, the completion routine being passed the response.

8. A method for asynchronously performing an operation on an object via a request by a client computer application to a server computer application, the method comprising the steps of:

obtaining an object reference to represent the object;

creating a proxy handle to represent the object reference;

calling the object with a stub function in the client application, the stub function being passed the proxy handle, an input parameter of the operation, and a client computer memory address to a completion routine in the client application;

transmitting the input parameter to the server application via an execution environment accessible to the client application and server application based upon the proxy handle;

implementing the operation on the object in the server application, the implementation including a response to the client application;

transmitting the response to the client application via the execution environment;

calling the completion routine in the client application, the completion routine being passed the response.

9. The method for performing an operation on an object, as recited in Claim 8, wherein the object reference is obtained from an initialization routine in the client application.

10. The method for performing an operation on an object, as recited in Claim 8, wherein the object reference is obtained from a disk file.

11. The method for performing an operation on an object, as recited in Claim 8, wherein the object reference is obtained from a previous object call.

12. The method for performing an operation on an object, as recited in Claim 8, wherein the step of implementing the operation on the object is performed asynchronously.

13. The method for performing an operation on an object, as recited in Claim 12, wherein the step of asynchronously implementing the operation on the object further comprises the steps of:

calling an asynchronous function from within the method, passing a call identifier and a response function address to the asynchronous function;

calling the response function from the asynchronous function, passing the call identifier to the response function; and

responding to the object call based upon the identifier.

14. A method for performing an operation on an object, the method comprising the steps of:

calling a method function to perform the operation, the function being passed an identifier to identify the object being called;

calling an asynchronous function from within the method function, passing the identifier to the asynchronous function; and

responding to the object call based upon the identifier.

15. The method for performing an operation on an object, as recited in Claim 14, further comprising the steps of:

allocating a portion of the server computer memory to handle the object call; and

deallocating the portion of the server computer memory after responding to the object call.

16. A computer program product, comprising:

a computer useable medium having computer readable code means embodied therein for performing an operation on an object via a request from a client computer application to a server computer application, the computer readable program code means comprising:

software means for obtaining an object reference to represent the object;

software means for calling the object with a stub function in the client application, the stub function being passed the object reference, an input parameter of the operation, and a client computer memory address to a completion routine in the client application;

software means for transmitting the object reference, input parameter, and completion routine address to an execution environment accessible to the client application and the server

application;

software means for transmitting the input parameter to a method in the server application via the execution environment based upon the object reference;

software means for implementing the operation on the object in the server application, the implementation including a response to the client application;

software means for transmitting the response to the client application via the execution environment;

software means for calling the completion routine in the client application, the completion routine being passed the response.

17. The computer program product, as recited in Claim 17, wherein the software means for causing one of the first computer and a second computer to implement the operation on the object further comprises:

software means for calling an asynchronous function from within the server application method, passing a call identifier and a response function address to the asynchronous function;

software means for calling the response function from the asynchronous function, passing the call identifier to the response function: and

software means for responding to the object call based upon the identifier.

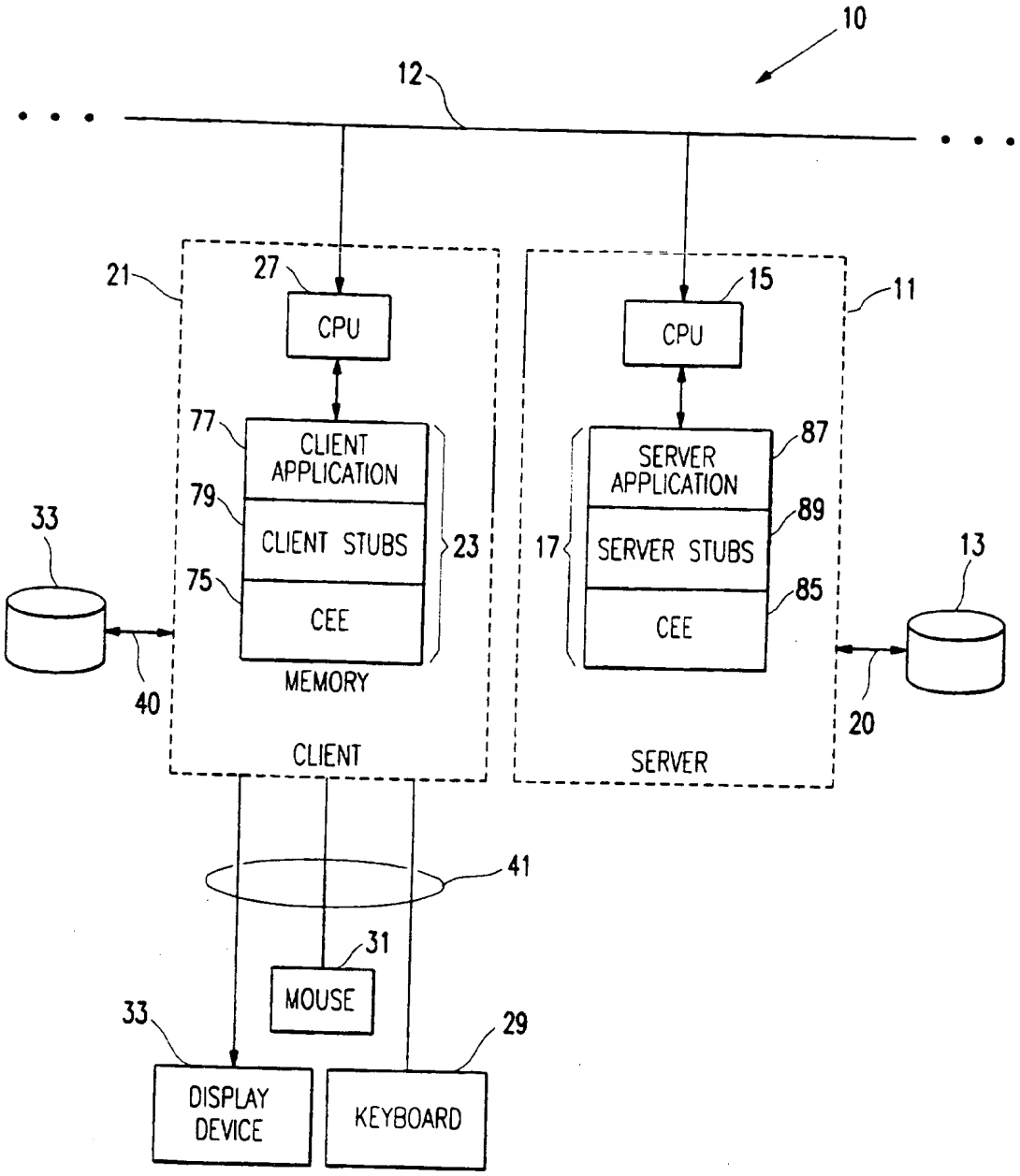


FIG. 1

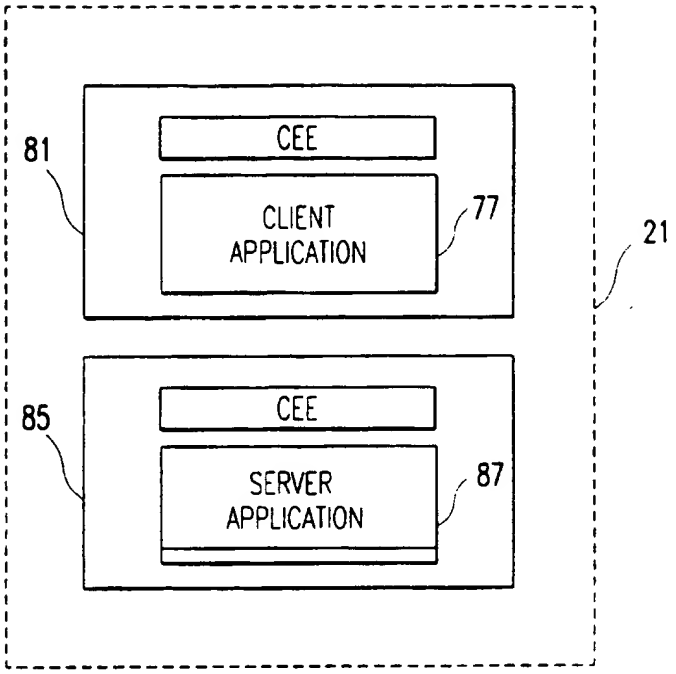


FIG. 2A

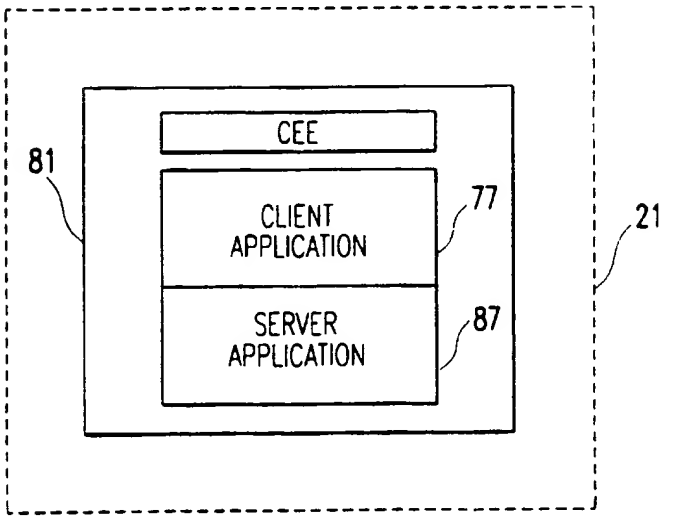


FIG. 2B

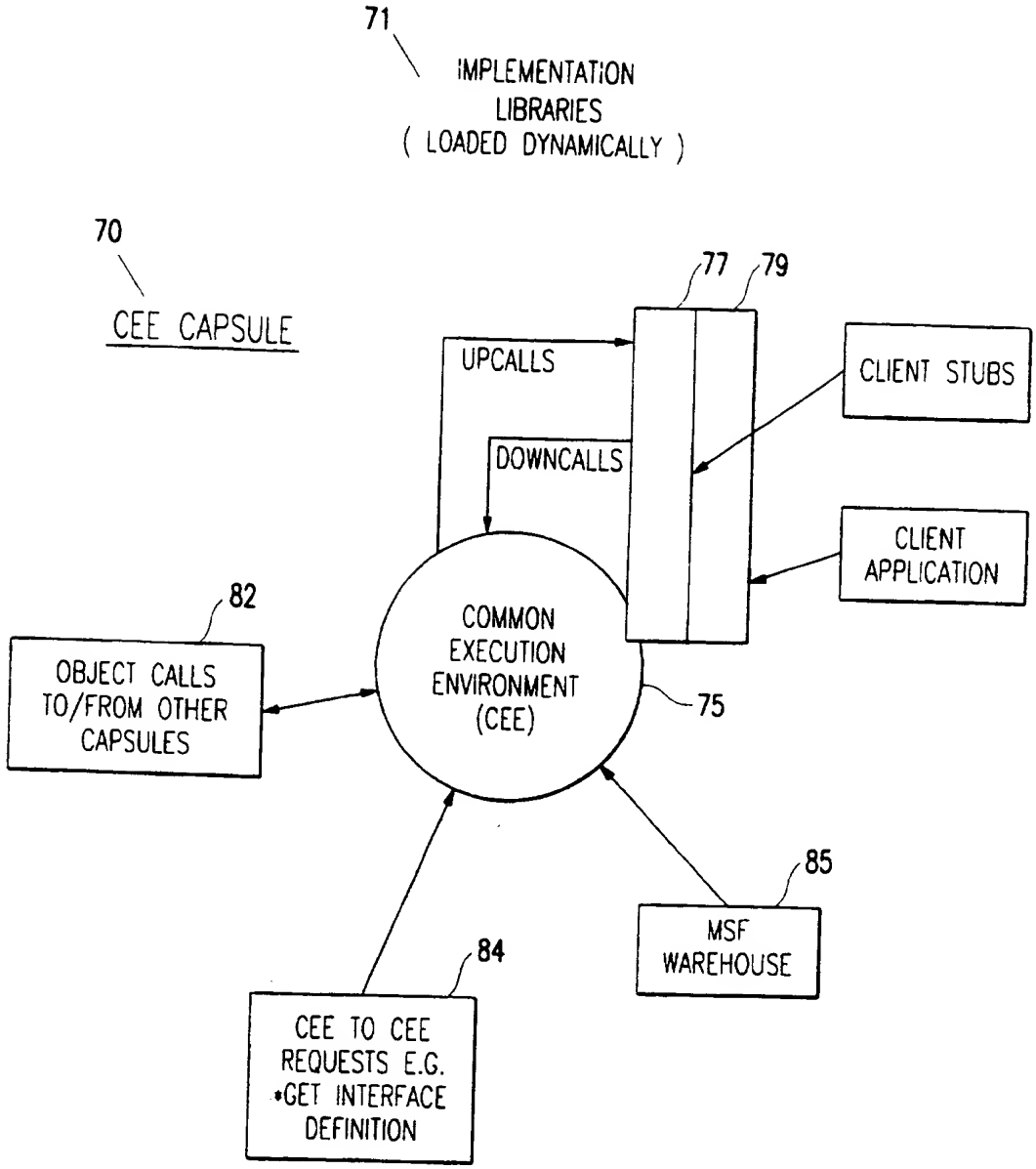


FIG. 3

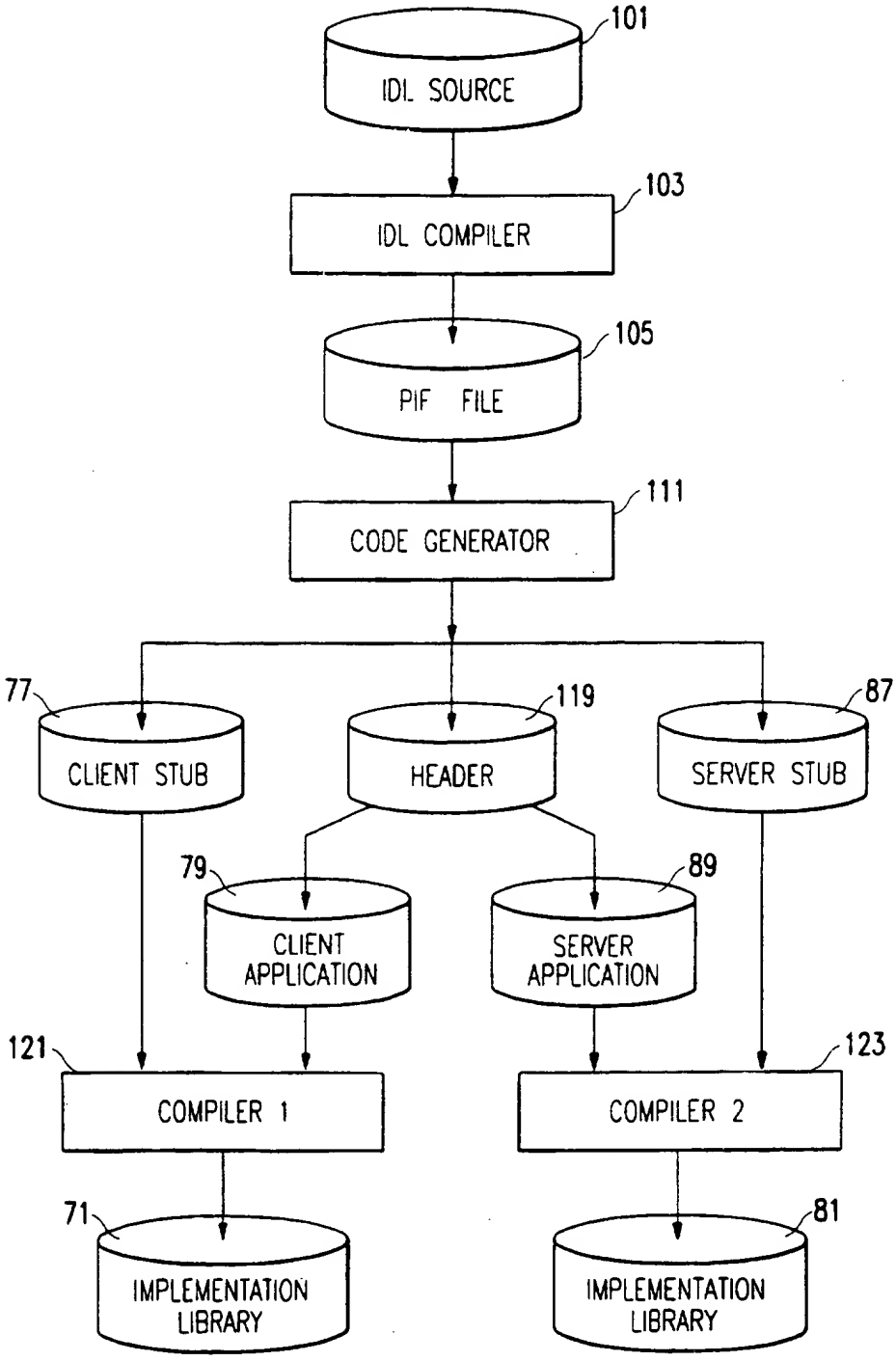
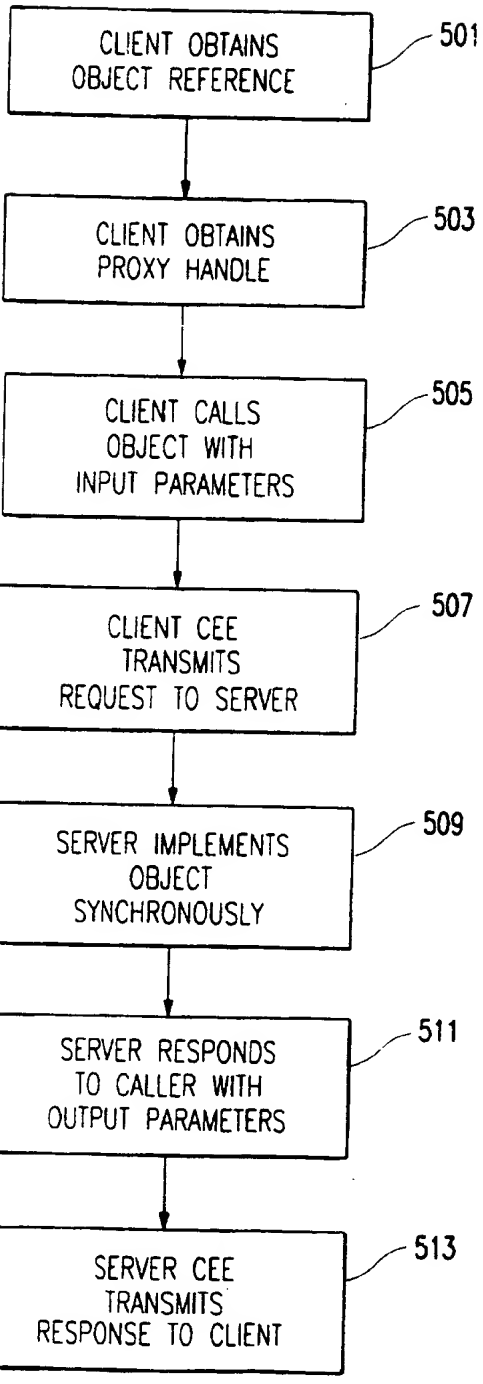
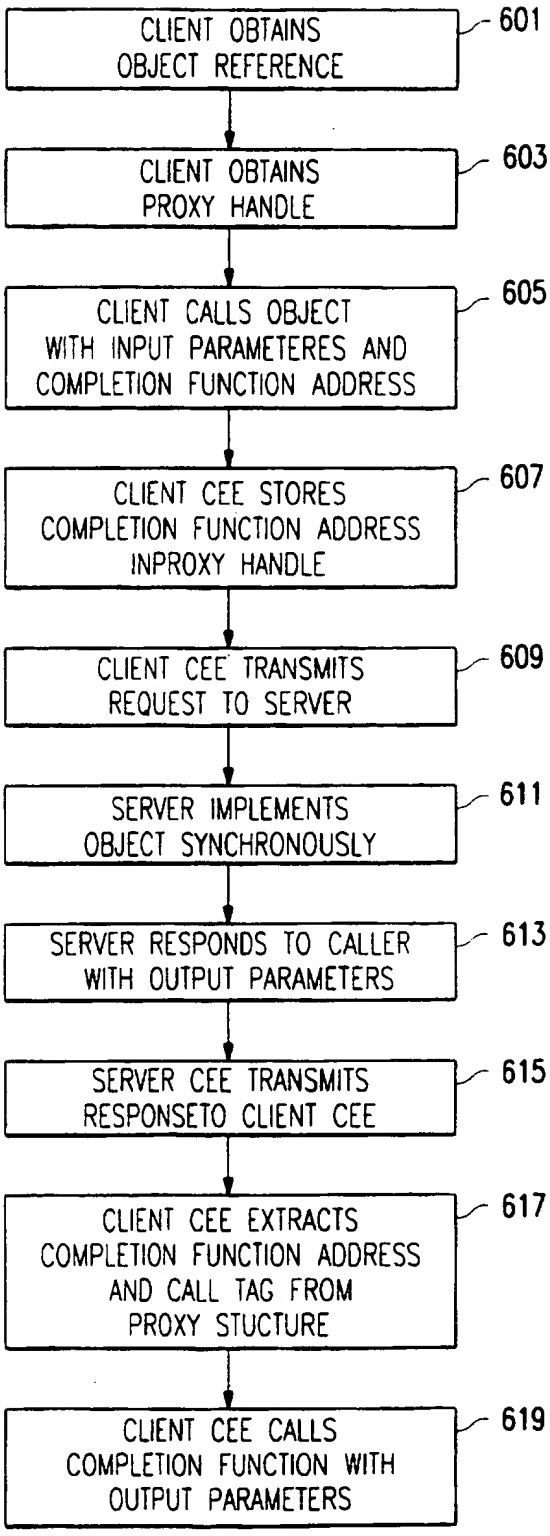


FIG. 4



SYNCHRONOUS OBJECT CALL/
SYNCHRONOUS IMPLEMENTATION

FIG. 5



ASYNHRONOUS OBJECT CALL/ SYNCHRONOUS IMPLEMENTATION

FIG. 6

```
#include"intf.h"                                              701

Client Init([Client Init parameters])                        702
        {
        sts=CEE_CFG_GET(Objref,&objref);                     703
        sts=TIM_ifc_(&intf);                                 704
        sts=CEE_PROXY_CREATE(objref,&intf,,&proxy);          705
        sts=CEE_PROXY_CREATE (objref,intf,&proxy1);          706
        sts=TIM_NOW_pst_(&proxy,LOCAL_TIME_TAG_GotTime,LOCAL); 707
        sts=TIM_NOW_pst(&proxy1,GMT_TAG,GotTime,GMT); }      708

GotTime (                                                    709
        In      cmptag_
        In      *exception_
        In      timeStr)
        {
        switch(cmptag )                                      710
                caseLOCAL_TIME_TAG                           711
                    printf("Local time is &%s\n",timeStr);   712
                    break;
                case GMT_TAG:                                713
                    printf("GMT is%s\n",timeStr);            714
                    break;
        }
```

FIG. 7

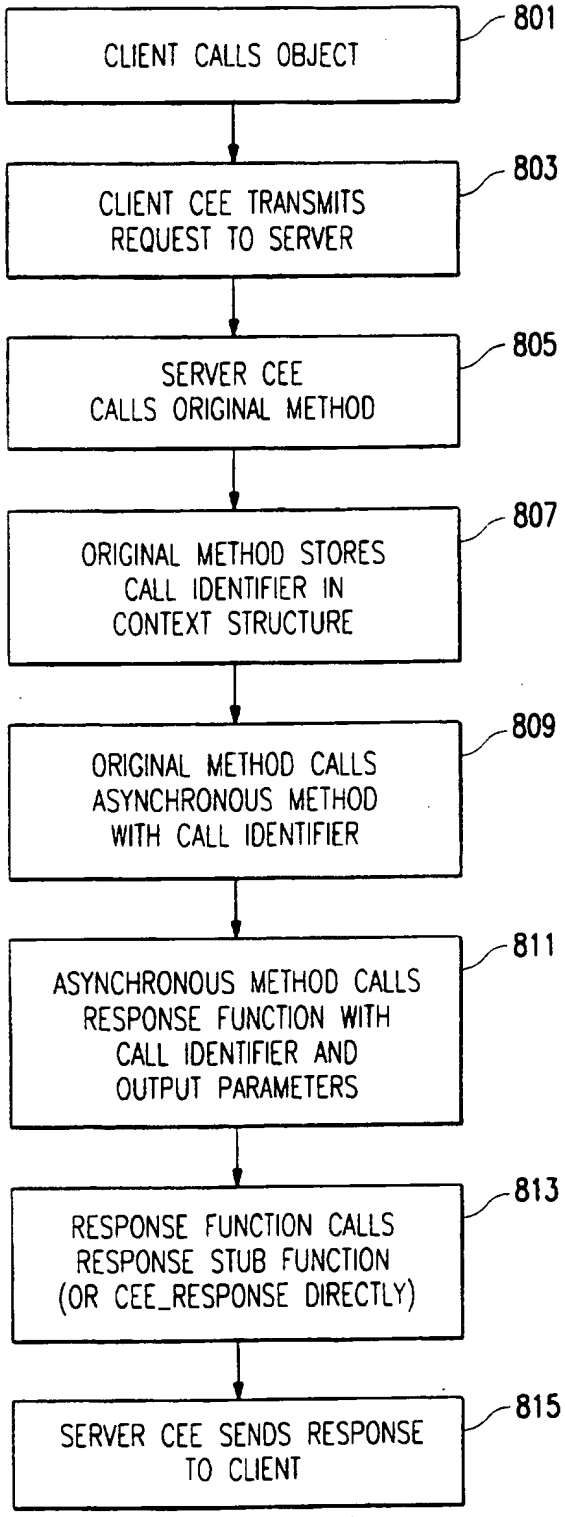


FIG. 8

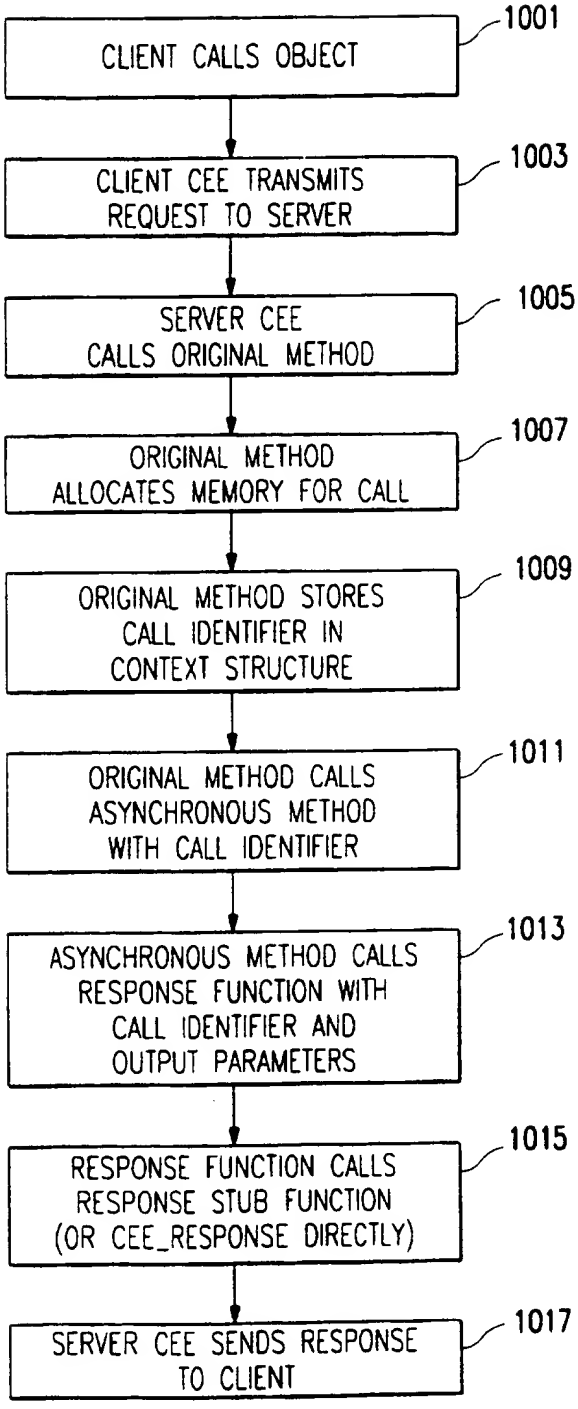
```
ImplInit ([ImplInit parameters])                                    901
{
        sts=TIM_ifc_(&intf);                                        902
        sts=CEE_IMPLEMENTATION_CREATE (&intf,implementationHandle);  903
        ssts=TIM_Now_ams_( implementationHandle, NowMethod);        904

NowMethod(
        objtag,
        call_id,
        timeZone)
{

        timeptr=(timeZone==GMT) ? gmtime (&t):localtime(&t);        905
        strftime(context->timeStr,sizeof(context->timestr),         906
        "%H"%M:%S",timeptr);                                        907
        context_callId=*call_id_;                                   908
        sts=CEE_TIMER_CREATE(1,0,TimerExpired,context,&timerHandle); 909
}

TimerExpired(context)
{
        sts=Tim_Now_Res_(&context->callId,&exeption,context->timestr); 910
}
```

FIG. 9



ASYNCHRONOUS IMPLEMENTATION WITH MEMORY ALLOCATION

FIG. 10

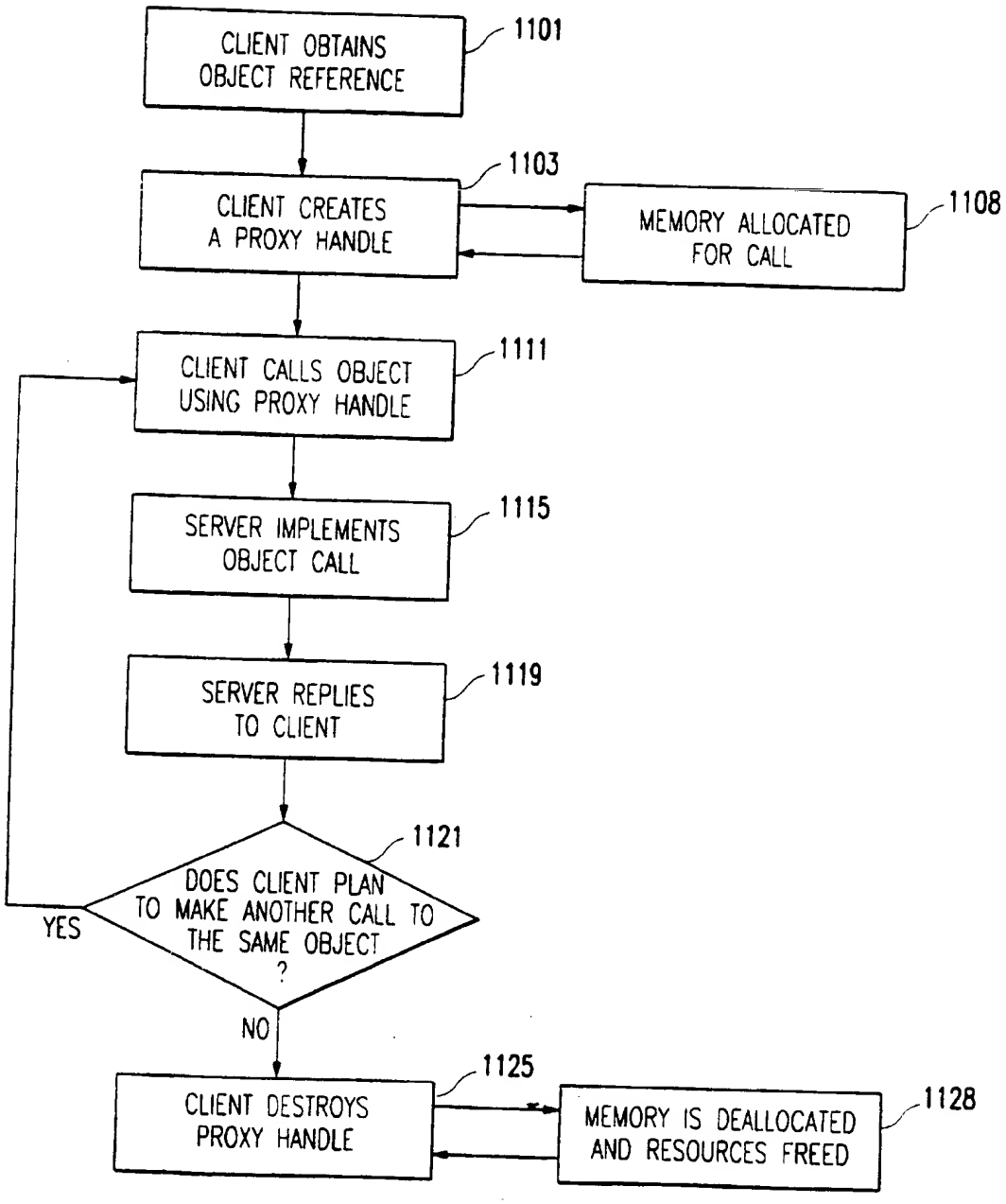


FIG. 11

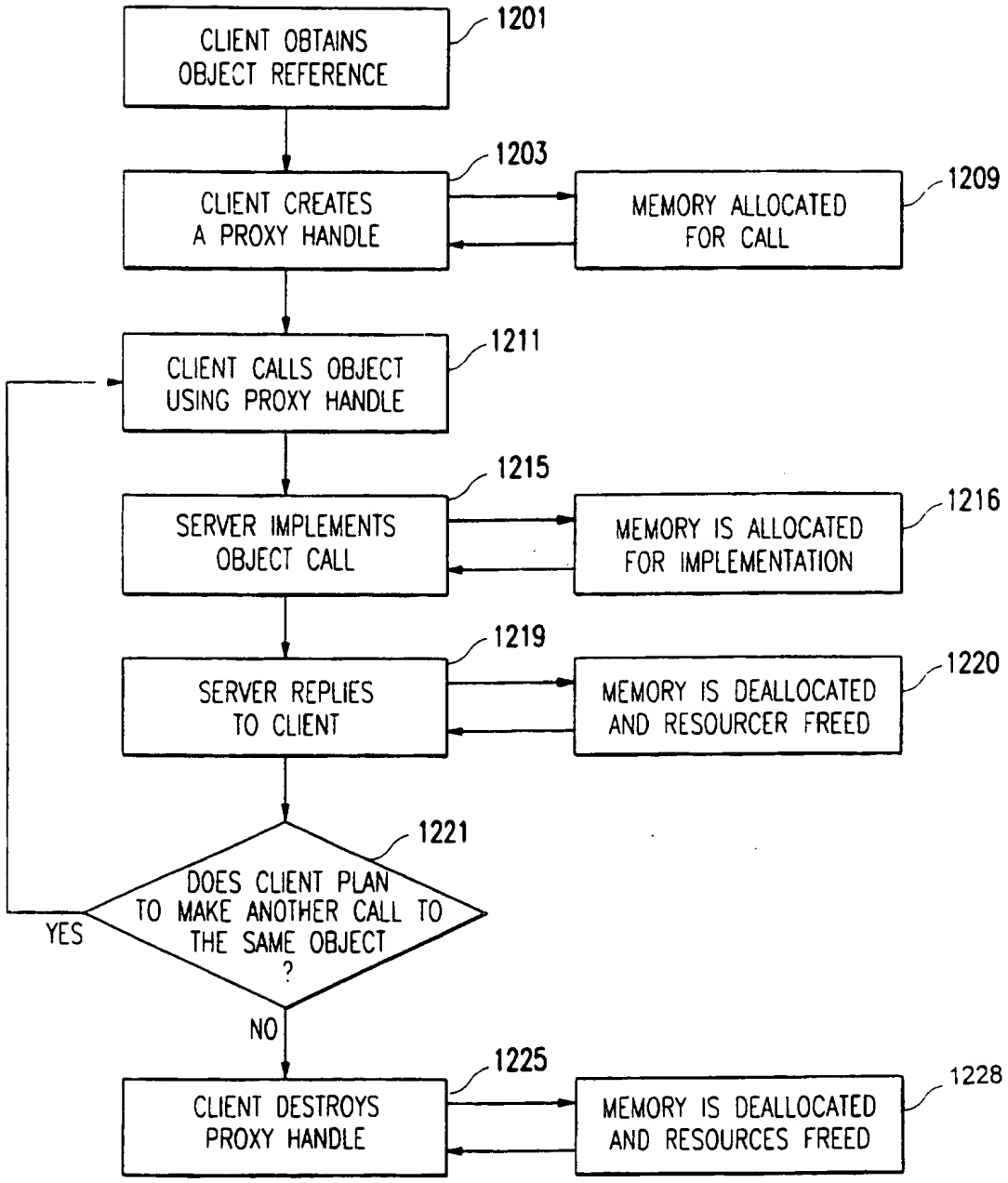


FIG. 12

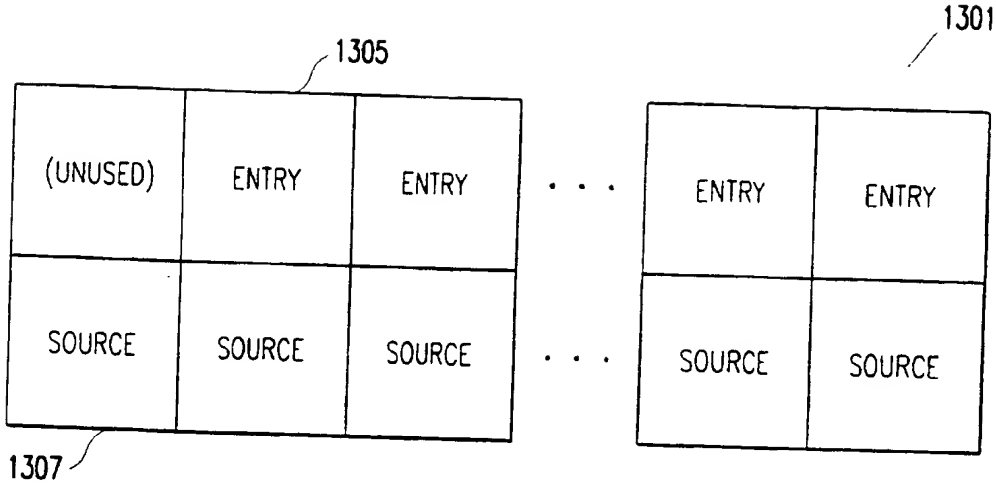


FIG. 13

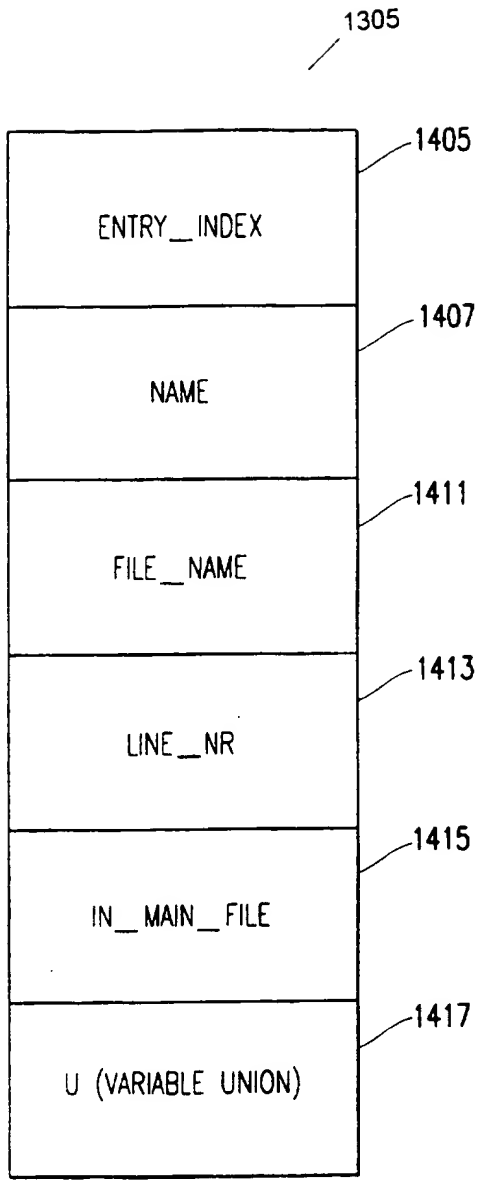


FIG. 14

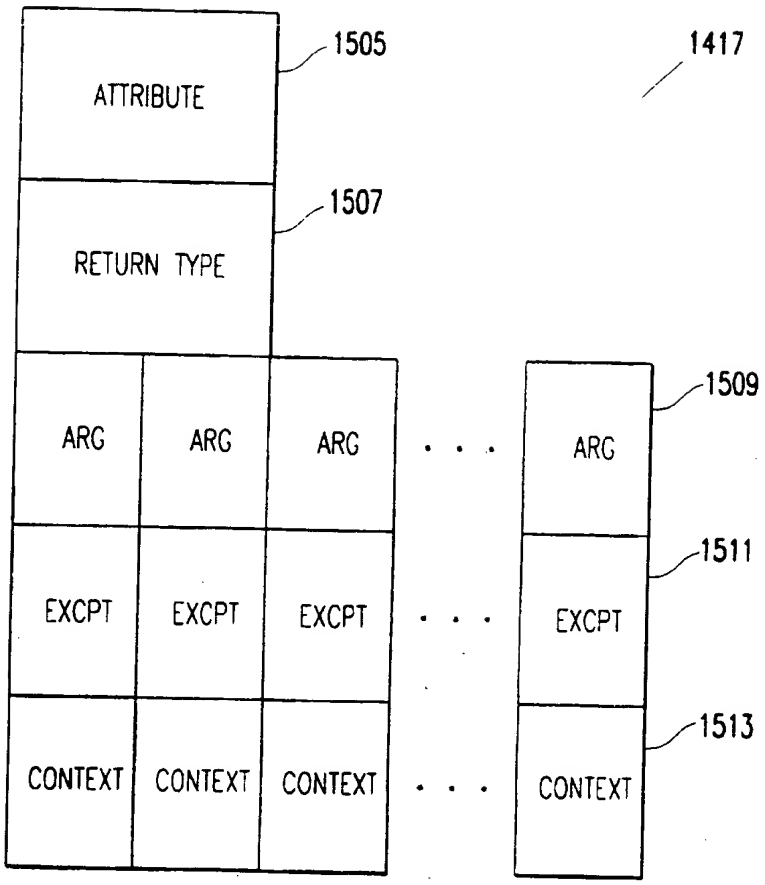


FIG. 15

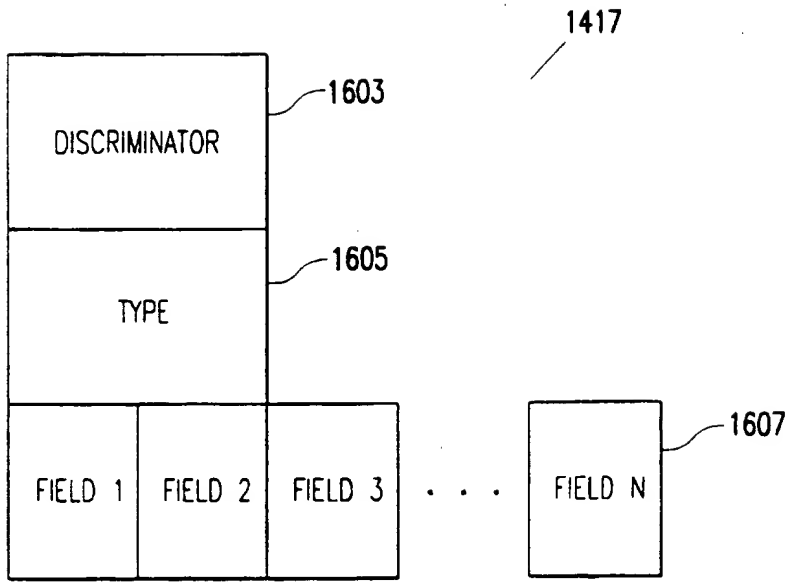


FIG. 16

INTERNATIONAL SEARCH REPORT

Interna al Application No
PCT/US 97/11879

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	MENON S ET AL: "OBJECT REPLACEMENT USING DYNAMIC PROXY UPDATES" PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON CONFIGURABLE DISTRIBUTED SYSTEMS, 1 January 1994, pages 82-91, XP002004310 see page 83, left-hand column, line 35 - page 84, left-hand column, last line see page 85, right-hand column, line 25 - line 45 see page 86, left-hand column, line 12 - right-hand column, last line --- -/--	1-17

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

2

Date of the actual completion of the international search	Date of mailing of the international search report
25 November 1997	03. 12. 97
Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Authorized officer Fonderson, A

Form PCT/ISA/210 (second sheet) (July 1992)

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	ANANDA A L ET AL: "ASTRA-AN ASYNCHRONOUS REMOTE PROCEDURE CALL FACILITY" INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, ARLINGTON, TEXAS, MAY 20 - 24, 1991, no. CONF. 11, 20 May 1991, INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, pages 172-179, XP000221855 see the whole document ---	1-17
A	CHATTERJEE A: "FUTURES: A MECHANISM FOR CONCURRENCY AMONG OBJECTS" PROCEEDINGS OF THE SUPERCOMPUTING CONFERENCE, RENO, NOV. 13 - 17, 1989, no. CONF. 2, 13 November 1989, INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, pages 562-567, XP000090924 see the whole document ---	1-17
A	IBM: "SOMOBJECTS DEVELOPER TOOLKIT USER GUIDE, VERSION 2.1 (CHAPTER 6)" October 1994 , IBM , US XP002047926 see page 6-24, line 24 - line 30 see page 6-27, line 5 - page 6-30, line 22 see page 6-48, line 41 - line 58 see page 6-66, line 35 - page 6-67, line 5 see page 6-72, line 45 - page 6-73, line 12 ---	1-17
A	US 5 247 676 A (OZUR MARK C ET AL) 21 September 1993 see the whole document -----	1,5-8, 12-14, 16,17

2

INTER..ATIONAL SEARCH REPORT

information on patent family members

Interr ..al Application No
PCT/US 97/11879

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5247676 A	21-09-93	US 5430876 A	04-07-95